Prioritizing Tasks in Code Repair: A Psychological Exploration of Computer Code

Gene M. Alarcon
Air Force Research Laboratory
Wright Patterson AFB, OH
gene.alarcon.1@us.af.mil

Sarah A. Jessup
Air Force Research Laboratory
Wright Patterson AFB, OH
sarah.jessup.ctr@us.af.mil

David W. Wood
General Dynamics Information
Technology, Dayton, OH
david.wood@gdit.com

Tyler J. Ryan
General Dynamics Information
Technology, Dayton, OH
tyler.ryan@gdit.com

August Capiola
Air Force Research Laboratory
Wright Patterson AFB, OH
august.capiola.ctr@us.af.mil

Abstract

The current study explored the influence of task prioritization on how computer programmers reviewed and edited code. Forty-five programmers recruited from Amazon Mechanical Turk downloaded and edited a computer program in C#. Programmers were given instructions to review the code and told to prioritize either the reputation, transparency, or performance aspects of the code, or were given no prioritization instruction. Code changes and remarks about their changes to the code were analyzed with a between-within multivariate analysis of variance. Results indicate prioritizing an aspect of the code leads to increased performance on that aspect, but with deficits to other aspects of the code. Managers may want programmers to prioritize certain aspects of code depending on the stage of development of the software (i.e., testing, rollout, etc.). However, managers should also be cognizant of the effects task prioritization has on programmer perceptions of the code as a whole.

1. Introduction

Computer code has become an integral part of modern society. Code relates to almost every aspect of our lives ranging from high-risk (e.g., health care databases, credit reporting agencies) to low-risk contexts (e.g., online gaming, cellphone storage). Despite the pervasiveness of code, little is known about how software programmers evaluate the code they reuse or repair, and the psychological processes that influence their subsequent decisions. Recent hacks to the Office of Personnel Management [1] and Equifax [2] illustrate the costs of using suboptimal code, as their architectures included bugs and security flaws. Most vulnerabilities found in code are legacy issues from previous iterations, indicating programmers ignored (or were not aware of) the issue in the code despite extant literature detailing these vulnerabilities [3]. Vulnerabilities such as memory violations (e.g., buffer overflows), input validation errors (e.g., SQL injection), privilege confusion (e.g., FTP bounce), or side channel attacks (e.g., timing attacks) are preventable assuming the code is properly vetted. The question remains: how do managers enable programmers to notice these vulnerabilities in the code and repair them? Psychological theories may help to explain why some programmers spot vulnerabilities in code, while others miss them altogether. Resource theories on task prioritization, such as multiple resource theory [4], posit that people have a limited supply of cognitive resources that when over tasked affect subsequent performance in tasks that are (or are not) prioritized. Prioritizing aspects of code repair may affect repairs and subsequent reporting of repairs.

RQ1: *Does task prioritization affect the types of changes made to code?*
RQ2: *Does task prioritization affect the types of remarks about the changes to code?*

1.1. Code Review

Code has become a ubiquitous aspect of society, but this has come at a cost. Programmers are hard-pressed to meet deadlines for large projects and must review code quickly because of the increasing demands for software development and updates. When software is first released, bugs and security flaws within the software can have large scale implications. For example, the Heartbleed issue found in 2014 allowed hackers to call back account information for numerous accounts, necessitating customers change their passwords once the vulnerability was fixed [5, 6, 7]. The patch was a minor one [8], in that it only required a change to one line of code. Specifically, the patch prevented buffer over-read, and many in the industry were surprised that they did not notice the vulnerability beforehand [9]. However, this issue is not unique to

URI: https://hdl.handle.net/10125/60090
ISBN: 978-0-9981331-2-6

Heartbleed. A recent article noted that 80-90% of vulnerabilities in software are legacy issues [3].

Issues such as the ones detailed above have led researchers to explore how programmers evaluate code. Researchers in computer science have utilized eye-tracking technologies and found that programmers who took longer to scan the code demonstrated better defect detection compared to programmers who took a shorter amount of time to scan the code [10]. Albayrak and Davenport [11] found aspects of the code such as indentation and naming defects influence false positive rates when inspecting code. Research on code review has also illustrated factors such as patch size, bug priority, and the organization itself can influence whether patches are accepted and how long it takes for them to be accepted [12]. However, little research to date has explored the psychological underpinnings of code review from the perspective of the programmer.

Psychologists have also gained an interest in the attributes of code that lead programmers to reuse code. Alarcon et al. [13] performed a cognitive task analysis (CTA) and found that three factors influence perceptions of code: reputation, transparency, and performance. *Reputation* is defined as how the code is assessed based on external data, such as the source from which the code originates (e.g., a coworker, a website) and the information available about the code (e.g., reviews, number of users, etc.) often found on external websites. *Transparency* is defined as how well the user understands the code. This can include aspects such as the readability and organization of the code, as well as comments throughout the code. Lastly, *performance* is defined as the perceived capacity of the code. Performance comprises aspects such as efficiency, resiliency, and flexibility of the code. The authors also noted the factors are not necessarily orthogonal, but instead may have conceptual overlap. For example, code that is efficient and concise may contribute to both perceptions of performance and transparency. Although Alarcon et al. [13] were focused on trust in and reuse of code, these three factors add insight into how programmers evaluate code. These factors, in part, compose the psychological foundation of how programmers perceive code and may influence how programmers review code [14].

1.2. Task Priority

Cognitive resource theories in psychology have posited individuals have a limited pool of cognitive resources [4, 15]. Therefore, people must often control their distribution of attention towards tasks they find particularly important [16, 17, 18]. Multiple resource theory [4] states an individual can allocate attention to multiple tasks at a time, but at a cost to performance. Conversely, people can focus attention on a single task, which may result in a lack of awareness for other events in the environment. For example, drivers may not notice an obstacle in the road while attempting to send a text message. Unusual stimuli in the road should normally attract attention, but if a different task is prioritized (i.e., texting a friend) the shift in attention may not occur leading the driver to crash [19]. Put simply, people have a limited capacity of resources, necessitating prioritization of those resources which ultimately affects performance outcomes in the real-world.

People can be instructed to prioritize one task over another [20]. In experimental settings, participants are typically instructed to complete a primary task (i.e., a task to prioritize) and a secondary task (i.e., a task that is *not* prioritized) simultaneously. When the primary task is prioritized, performance increases on this task while secondary task performance decreases [20, 21]. Instructing a worker to prioritize one task over another can lead to a performance tradeoff, despite no changes in the actual tasks [20, 21, 22]. Additionally, the degradation in secondary task performance can be used to quantify the resources allocated towards the primary task [23].

Concurrent tasks, requiring overlapping resource requirements, are cognitively demanding. However, if the resources are from different modalities, multitasking may be possible. For example, an air traffic controller can be expected to acknowledge vocally while still monitoring a visual space, as each task does not demand the same resources [4]. In contrast, a driver trying to text and drive experiences interference as both the texting and driving tasks require overlapping perceptual and cognitive resources. Code review is a unique task in that it requires both cognitive comprehension and mental rotation for adequate performance.

1.3. Task Priority in Coding

Software programming provides a unique context to explore task priority. Applied researchers have noted that studies in health care (e.g., [24]) and driving (e.g., [25, 26]) lack ecological validity, as researchers do not want to put a participant's life at risk during the experiment (e.g., driving, hospital situations). Further, past research has not found a straightforward relationship between task priority and performance. For example, Waller and colleagues discussed how contextual factors are key in making task-prioritization more or less important for performance [27, 28].

In the code review process, programmers may be instructed to prioritize one task over another. Programmers can allocate specific resources to a specific aspect of the review and increase their performance on that aspect, but this may result in

programmers not noticing other aspects of the code. Focusing on a specific aspect of code will tax cognitive resources, as the programmer will try to find issues that are related to those aspects they are to prioritize. This allocation of resources towards the primary task should increase performance on the aspect of code prioritized, but programmers should miss other issues as working memory is narrowed on the primary task. Consider one piece of code given to multiple programmers: one programmer may be instructed to look for vulnerabilities, whereas as another programmer may be instructed to look for functional defects. These instructions can come from either their supervisor (e.g., "fix this memory leak," "search for vulnerabilities") or from the job position itself (e.g., security specialist) [12]. Therefore, it is important to consider how task prioritization influences subsequent programmer performance.

The software review context provides an ecologically valid environment to investigate the influence of task priority on performance. An experimenter can provide a programmer with code for review and repair much like a manager provides a task to an employee. This allows for multiple aspects of the code to be manipulated, and the participants can be instructed to prioritize different tasks. The same code can be used for all participants to determine if task prioritization improves performance in that task. For example, in code review there may be several different issues such as vulnerabilities, breaks with coding conventions, and unnecessary processes, but only focusing on breaks in coding conventions could result in neglecting vulnerabilities.

1.4. Amazon Mechanical Turk

Amazon Mechanical Turk [29] is an online crowdsourcing website in which individuals (or "workers") can participate in research studies or complete tasks for monetary compensation. Businesses or individuals can post jobs or tasks, known as Human Intelligence Tasks (HITs), that computers are currently unable to perform [30]. MTurk has proven itself to be a valuable resource for psychological research, allowing access to a large diverse subject pool with little cost to researchers compared to classical psychological studies [31, 32, 33]. Additionally, researchers have demonstrated data collected from MTurk offers comparable results to classical psychological studies [34, 35].

The majority of psychological research utilizing MTurk has been on self-reports. Although self-reports are a valid psychological measure of internal cognitive states, behaviors have largely been ignored in research conducted on MTurk, and to a larger degree psychology [36]. The MTurk platform has the ability to request tasks such as writing product descriptions, translating from one language to another, and transcribing audio files. One such task that can be performed via MTurk is code review and repair. In a study on perceptions of code trustworthiness, Alarcon et al. [37] were able to replicate their main effects of code manipulations on trustworthiness perceptions found in an in-person sample on MTurk, indicating programmers are available on the platform. The platform makes it possible to have programmers review, edit, and repost code to accurately explore behaviors via MTurk without knowing they are in an experiment.

We utilized the factors from Alarcon et al.'s [13] CTA to assign participants a task priority (i.e., reputation, transparency, or performance) when reviewing code. This is the first study the authors are aware of that has attempted to manipulate task prioritization in code repair and the first to use MTurk for such a task. We hypothesize programmers will make more changes to the code according to the task they are assigned to prioritize compared to tasks they have not been assigned to prioritize.

2. Method

2.1. Stimuli

Code for an image filter was created as stimuli for the current study. We did not utilize a previously developed program for degradation because the participants may have been able to find the code online and simply upload the non-degraded version. Research has demonstrated MTurk participants are more likely to use the internet to find answers, even with no incentive for correct responses [31]. As such, we wanted to ensure participants could not find the non-degraded code online and repost it. Participants in all conditions received the same image filter stimuli with the same degradations. The two main classes in the program used for analysis totaled 442 lines of code written in the C# programming language. C# was utilized because it was the primary language of the programmer who created the stimuli in the experiment. The code stimuli are available to view online (https://www.github.com/PerfLogistics/ImageFilter).

The code was degraded according to the three factors described by Alarcon et al. [13], namely reputation, transparency, and performance. Several manipulations were guided by previous studies that also degraded source code [37, 38]. Reputation manipulations to the code entailed inclusion of unused dependencies. Transparency manipulations to the code included removal of meaningful comments, leaving in code that is commented out, using unintuitive variable

names, and absent or poor use of whitespace and indentation. Finally, performance manipulations to the code entailed code that had no meaningful function, poor or absent error handling, and code that unnecessarily accessed the system's BIOS.

2.2. Procedure

The study was posted to MTurk as a task to be completed (a HIT). To ensure ecological validity, a fake company name was created to ensure participants were unaware they were participating in a psychological experiment. Requirements for the study were that participants know the C# programming language, submitted code changes must compile, and participants had to have a GitHub account or the ability to create one so that they could access the code. Participants were able to view the code prior to accepting the HIT by following the link to the code on GitHub. Once participants accepted the HIT, they had 24 hours to complete the task of cleaning and repairing the code.

Participants completed the HIT in one of four conditions: 1) control, 2) reputation, 3) transparency, or 4) performance. All conditions contained the following description, "[Company Name] is looking for assistance with completing and correcting computer code created by one of our students. Participants will be compensated with $20.00 for their participation. $50.00 will be awarded to the best code presented and chosen to be implemented within our work." The bonus was utilized to ensure participants were adequately incentivized to perform the task well. A control condition was included to compare the experimental conditions to general code review instructions. The control condition stated, "After clicking on the GitHub link you will find code created by one of the students working in our office. The program is meant to be used as an Image filter, though we see it has several bugs. We'd like for you to review the code and fix any errors you find in it." No other directions were given. Each of the conditions contained the control condition statement.

In each experimental condition, an additional statement focused on one of the three factors based on the CTA by Alarcon et al. [13]. The reputation condition included the statement, "The code seems to be using several different references and we are unsure what they do, and if they are completely necessary. For your task, we need you to review the code and how it uses the reference and remove any redundancies. In addition, if you know of a reference online that could be used in place of what the student used, add and implement that into the code." The transparency condition included the statement, "We are having issues reading and understanding the code that the student created. The code isn't completely commented and seems to be all out of place. The code will be used for a larger project and will be public to others in our field, so the code should look professional. We need you to comment the code and clean up formatting where you see needed." Lastly, the performance condition included the statement, "The code is having issues running correctly. We aren't sure if our student used the best methods for the image processing. In addition, the GUI is very crowded and we'd like it condensed into a list box. We would like for you to find any performance issues in the code and correct them, as well as also correcting the GUI." Participants were instructed to review and repair the code according to one of four conditions. For example, if a participant was assigned to the transparency condition, then reviewing and repairing the transparency manipulations was their primary task directive but they still received the control condition message to correct and complete the code in general. It should be noted that because the stimuli was the same in every condition, participants could make reputation, transparency, and performance changes and remarks, regardless of the condition to which they were assigned. For example, in the transparency condition participants were asked to fix the transparently issues, but the same performance issues in the performance condition were also in the transparency and reputation conditions.

We posted each condition separately so that only one condition was running on MTurk at a time. Participants who accepted the posted HIT were excluded from subsequent posted HITs. Participants accepted one of four conditions (control, reputation, transparency, or performance), depending on what was available when they were online, and participants downloaded the code from the GitHub website. After completing the task, participants copied the repaired code as a text document into a response field in MTurk. Next, participants were given a chance to describe their changes to the code and why we should choose their code for implementation. Participants were not required to describe their changes to received compensation. After completing data collection, two experienced programmers reviewed the code and determined the code with the best code repair for each of the four conditions. In each condition, the MTurk participant that cleaned and repaired the code best was then credited with the bonus money (i.e., $50.00 USD) through MTurk.

2.3. Participants

A total of 106 participants were recruited from MTurk. Participants were paid $20.00 USD to clean and repair a piece of code (see Stimuli). The participation cost per participant is much larger than traditional MTurk studies. However, we felt the task justified the pay as the task would take about an hour to complete and the participants needed a particular skill, namely programming. We excluded 61 participants because

either they did not make any changes to the code, or the participants submitted work from another online repository instead of changing the code as directed, failing to follow the task instructions. This left a total of 45 participants for analyses across the four conditions [control N = 10 (13 rejected), reputation N = 12 (13 rejected), transparency N = 11 (21 rejected), performance N = 12 (14 rejected)]. No demographics were collected on participants as it would allude to the experimental nature of the study. In addition, we were concerned that participants may inflate their experience to get the $50 bonus. We did not verify if the participant knew C#, as a requirement of being compensated was that the code compiled. As such, if participants' changes compiled then it was assumed they had at least a functional knowledge of programming in the C# language. Any participant that signed up for the study under one condition was blocked from viewing the study using the qualifications tool regardless if they completed the task or not. This was done to ensure we did not collect data on the same participant several times. The study was overseen by the Air Force Research Laboratory institutional review board.

2.4. Data Cleaning

A researcher with a programming background used a utility tool to calculate the changes participants made to the code. This tool compared the differences between the original piece of code and the augmented version, allowing the researcher to determine what changes were made by each participant. For clarity in explaining our results, we refer to these changes as "*Updates*" in our model. The programmer who created the stimuli qualitatively coded the changes made by the participants from MTurk into one of three categories. Reputation *Updates* consisted of adding or removing libraries. Transparency *Updates* consisted of adding comments, removing comments, adding spaces between numbers and signs, changing the names of variables, and indenting. Lastly, Performance *Updates* consisted of enumerating, removing methods, adding methods, adding disposals, event handlings, switches, dictionaries adding try/catch, and deleting lines of code. Two additional programmers with no knowledge of the software replicated the binning of the *Updates* into the aforementioned categories. There were no changes made to the code that were not encompassed by these three categories. After coding the changes, the programmer then qualitatively coded the remarks MTurk programmers made describing their code into one of three categories: reputation, transparency, and performance. These *Remarks* served as manipulation checks and analyses were conducted on the *Remarks*. The *Updates* and *Remarks* were analyzed using multivariate analysis of variance (MANOVA).

The *Update* variable consists of all reputation, transparency, and performance changes for the individual. The *Remarks* variable consists of all reputation, transparency, and performance related remarks an individual made about their changes to the code. *Updates* and *Remarks* are nested within individuals. The data were structured as a repeated measures design. In other words, one participant had reputation changes, transparency changes, and performance changes as outcomes that were all part of the *Update* outcome variable. We structured the data this way to be able to account for individual differences in the outcomes. The within-subject variable consists of trustworthiness factors indicating the type of change (i.e., reputation, transparency, or performance), which we refer to as Categories of Changes.

3. Results

To address RQ1 and RQ2, a two-way mixed design multivariate analysis of variance (MANOVA) was run to determine the effect of Condition (between-subjects factor) and Categories of Change (within-subjects factor) factors on the number of *Updates* and *Remarks* (dependent variables). We analyzed the data against a null hypothesis that no significant differences between Condition groups exist and no significant differences between Categories of Change exist. Preliminary assumption checking revealed that data was not normally distributed, as assessed by Shapiro-Wilk test ($p < .05$). After applying a square root transformation, normality worsened. As such, we did not transform our data. There were univariate outliers as assessed by boxplot but no multivariate outliers, Mahalanobis distance ($p > .001$); we retained all cases. A Greenhouse-Geisser correction was applied when the sphericity assumption was not met.

A repeated measures MANOVA was conducted to test Condition and Category of Changes made on *Updates* and *Remarks*. Condition had a significant influence on both dependent variables (*Updates and Remarks*) [Pillai's $V = .45$, $F(6, 82) = 3.91$, $p < .01$, $\eta_p^2 = .22$, power = .99]. There was a significant difference in the Category of Changes in the dependent variables (*Updates and Remarks*) [Pillai's $V = .74$, $F(4, 38) = 27.64$, $p < .001$, $\eta_p^2 = .74$, power = .99]. Additionally, Condition moderated the effect of Categories of Change on the dependent variables [Pillai's $V = .895$, $F(12, 120) = 4.25$, $p < .001$, $\eta_p^2 = .30$, power = .99]. Both factors resulted in critical p-values less than the selected significance level, indicating the Condition varied significantly across groups and there were differences in Categories of Change. As both research questions were qualified by an interaction, we conducted simple effects analyses to determine the nature of the relationship. To

Table 1. Estimated Means and Standard Errors of Updates and Remarks (Standard Deviations are in Parentheses of Mean Totals)

Condition	Updates			Remarks		
	Reputation	Transparency	Performance	Reputation	Transparency	Performance
Control	.90 (.41)	18.00 (4.82)	8.30 (2.21)	.50 (.22)	1.20 (.76)	1.10 (.41)
Reputation	1.75 (.65)	29.08 (8.52)	2.75 (.698)	1.17 (.27)	1.92 (.58)	2.25 (.62)
Transparency	1.82 (.74)	62.09 (7.29)	4.00 (1.12)	1.55 (.62)	3.82 (.74)	2.45 (.64)
Performance	1.17 (.49)	13.42 (3.91)	9.17(1.51)	.33 (.14)	.83 (.37)	3.75 (.95)
Mean Total	1.42 (1.96)	30.51 (28.45)	6.00 (5.41)	.89 (1.27)	1.93 (2.30)	2.44 (2.47)

determine the interactions and simple effects, univariate ANOVA analyses were conducted. To minimize Type I error rates, the ANOVA significance levels were adjusted for each analysis per Cramer [39]. All univariate main effects ANOVA analyses significance level was adjusted to .025(.05/2).

3.1. Univariate Updates Analyses

The univariate main effect of Condition was significant for *Updates* [$F(3, 41) = 8.42$, $p < .001$, $\eta_p^2 = .38$, power = .99] indicating task prioritization influenced participant's *Updates*. The univariate main effect of Categories of Changes was significant for *Updates* [$F(1.08, 44.21) = 68.38$, $p < .001$, $\eta_p^2 = .63$, power = .99] indicating there were significant differences in the types of *Updates* the participants were making. The results of the ANOVAs were qualified by an interaction between the factors on *Updates* [$F(3.24, 44.21) = 12.23$, $p < .001$, $\eta_p^2 = .47$, power = .99]. Means and standard errors are illustrated in Table 1. To examine the univariate simple effects of Condition on *Updates*, the one-way ANOVA analysis significance level for simple effects was set to .006 (.025/4). We chose to use Dunnett's C to examine post-hoc tests.

First, we explored the influence of Condition within each Category of Changes for *Updates*. There was a significant simple effect of Condition on Transparency *Updates* [$F(3, 41) = 11.24$, $p < .001$]. Participants made more Transparency *Updates* in the Transparency Condition compared to the Control Condition (Mean difference = 44.09, $SE = 8.74$, $p < .01$) and Performance Condition (Mean difference = 48.67, $SE = 8.27$, $p < .006$). Also, there was a significant simple effect of Condition on Performance *Updates* [$F(3, 41) = 4.98$, $p = .005$], such that participants in the Control and Performance Conditions made more Performance *Updates* than the Reputation or Transparency Conditions. However, due to our stringent *p*-value cutoff in an effort to reduce Type 1 error, no post hoc analyses emerged as significant for the Performance Condition. The post-hoc tests for Reputation *Updates* were not significant. Figure 1 illustrates the results of the analyses.

Next, we explored the differences in Categories of Changes within each Condition. We examined the differences between the Categories of Changes in the *Updates* variable, and the repeated measures ANOVA analysis significance level was set to .008 (.025/3). There was a significant simple effect of Categories of Changes on *Updates* in the Control Condition [$F(2, 18) = 10.47$, $p = .001$, $\eta_p^2 = .54$, power = .99] such that participants in the Control Condition made significantly more Transparency than Reputation *Updates* (Mean difference = 17.10, $SE = 4.69$, $p = .005$) and more Performance than Reputation *Updates* (Mean difference = 7.40, $SE = 2.18$, $p = .008$). Additionally, there was a statistically significant simple main effect of Categories of Changes in the Transparency Condition [$F(1.04, 10.37) = 66.17$, $p < .001$, $\eta_p^2 = .87$, power = .99], such that participants made significantly more Transparency than Reputation *Updates* (Mean difference = 60.27, SE = 6.87, $p < .001$), and more Transparency than Performance *Updates* (Mean difference = 58.09, $SE = 7.55$, $p < .001$). The mean differences between Categories of Changes in the Reputation and Performance Conditions were not significant.

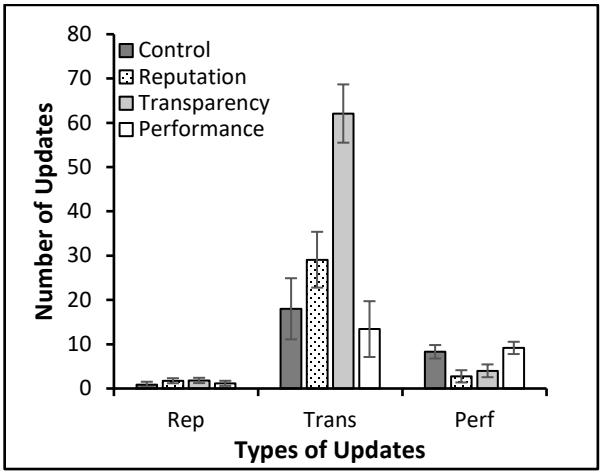

Figure 1. Programmer changes to code by condition.

3.2. Univariate Remarks Analyses

The univariate main effect of Condition was not significant for *Remarks* [$F(3, 41) = 2.52$, $p = .07$, $\eta_p^2 = .16$, power = .86], indicating task prioritization did not influence how they described the code. Means and standard errors are illustrated in Table 1.

The univariate main effect of Categories of Changes was significant on *Remarks* [$F(2, 82) = 9.87$, $p < .001$, $\eta_p^2 = .19$, power = .92], indicating participants made more remarks about some categories than others. However, these main effects were qualified by a significant interaction between Condition and Categories of Changes [$F(6, 82) = 3.97$, $p < .01$, $\eta_p^2 = .23$, power = .99]. As such we explored the differences in Categories of Changes within each Condition for *Remarks*. To examine univariate simple effects of Categories of Change on *Remarks*, the repeated measures ANOVA analysis significance level for simple effects was set to .008 (.025/3). There was a significant simple effect of Categories of Change on *Remarks* in the Performance Condition [$F(1.12, 12.95) = 10.11$, $p = .006$, $\eta_p^2 = .48$, power = .99], such that participants in the Performance Condition made significantly more Performance *Remarks* than Reputation *Remarks* (Mean difference = 3.42, *SE* = .95, $p = .004$). Results are illustrated in Figure 2. No significant differences were observed between any other conditions.

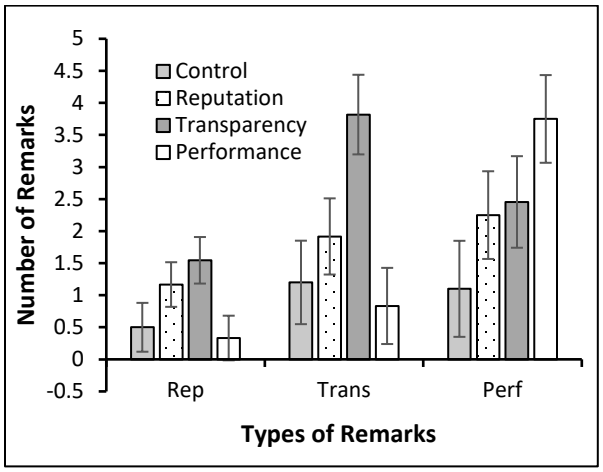

Figure 2. Counts of Remarks programmers made about changes to code by Condition and Type of Remark.

4. Discussion

Results from this study indicated task prioritization influenced the *Updates* the programmers made to the code and to a lesser degree, their subsequent *Remarks* about their changes to the code. Instructing participants to prioritize fixing a certain aspect of the code (i.e., reputation, transparency, or performance) had a direct influence on the type of changes made to the code in the transparency and performance condition. The current study is unique in that we included a control group and were able to compare prioritized performance differences with a non-prioritized condition in a real-world setting. As expected, prioritizing a task generally led to increased performance on the task, as can be seen in the transparency and performance changes to the code. However, this was not the case in the reputation condition. It may be that the reputation condition was not conceptually accurate, or had too much overlap with transparency. Indeed, when no other information about the author is available, the programmers may have assessed reputation via lack of transparency in the code.

Interestingly, there were no significant differences between the control condition and the performance condition on performance changes. When given no other guidance, performance is a key aspect of the code [37]. However, there were performance deficits when participants were instructed to search for reputation or transparency issues. The psychological resources allocated towards inspecting reputation or transparency issues may have distracted participants from finding the performance issues. However, even when instructed to prioritize a task, participants still conducted other changes to the code. Researchers have discussed the importance of context when examining the relationship between task priority and performance in occupational simulations [27]. These findings offer support that code use and reuse scenarios may provide a context where task prioritization is related to subsequent performance.

The findings in this paper are unique in that they focused on actual task performance as well as how the participant perceived the task. The results indicate that programmers will act in accordance with the task they were assigned to complete, and report doing so, as we would expect. However, it also illustrates the problem of task prioritization in that a supervisor asking a programmer to prioritize a task will also lead to a deficit in finding other deficiencies. This is particularly problematic for the transparency and reputation conditions in that it led to fewer performance changes.

An important aspect of the current study is inclusion of a control condition. Previous research on task prioritization rarely included a control condition when comparing performance on the primary and secondary tasks [20, 21]. We were able to compare conditions with different task priorities to determine if any performance increases were due to the conditions. Namely, all participants had the same stimuli (i.e., the referent code) and were able to make any *Updates* or *Remarks* they chose. In addition, participants had a significant amount of time to complete the task. Despite the abundance of time, participants still exhibited decrements in non-prioritized task performance when asked to prioritize another task. Additionally, previous research has

focused on time limits when conducting task prioritization research. It is interesting that task prioritization still had an effect, to some degree, when the participant had practically unlimited time resources (as they had 24 hours to complete fewer than 500 lines of code repair).

This study is the first of its kind to explore actual behaviors in programmer's changes to code in an ecologically valid context. Outsourcing code repair is becoming a popular alternative to repairing code in house [40]. The current study utilized this trend to collect behavioral data on the MTurk platform from individuals. This study indicates the platform can be used to address the call of more behavioral data in the sciences rather than self-report [36]. Additionally, the results should easily approximate to real-world settings as participants were unaware it was an experiment. This study offers a template for using MTurk, offering researchers another avenue of using MTurk rather than Likert-type responding to scales in mass.

4.1. Theoretical Relationships of Reputation, Transparency, and Performance

We utilized the constructs from Alarcon et al.'s [13] CTA. In their paper, the authors conceded the reputation, transparency, and performance constructs are not orthogonal. Of particular interest in the current study is the overlap of transparency and reputation. When participants were asked to prioritize transparency, there was no significant difference in the number of performance changes compared to the reputation condition. Similarly, those in the reputation condition performed the second most transparency changes compared to the transparency condition. Transparency and reputation may be two constructs that share considerable overlap in the programming community. Increasing the transparency of code may also increase reputation. Code that is disorganized, uses poor naming schemas, and is hard to read may be perceived as suspicious, thus decreasing the reputation assessment of the code and the programmer or source where from which the code was obtained.

From a multiple resource theory perspective, participants prioritized tasks which may have had overlap with other tasks that were not prioritized. For example, when evaluating the transparency of the code one might still evaluate the performance to some degree as the constructs are related and the same cognitive resources are being utilized. Thus, participants are not doing two things at once but rather one task has overlap with a task the participants were not prioritized to conduct. Wickens [4] stated tasks that tax the same resources will lower performance and accuracy, unless those tasks have conceptual overlap and facilitate each other. It may be that some aspects of each of the reputation, transparency, and performance conditions are necessary for other aspects and that none of the constructs exist independently, but are instead dependent on each other, to some degree, for code comprehension.

4.2. Limitations

The current study is not without limitations. One limitation of the current study is the small sample size. The current study has strong main effects of condition on code changes and post-hoc power analyses illustrated enough power to detect the effects found. However, when interpreting the interactions of type of change to the code (i.e. reputation, transparency, or performance) and the participant's condition, we experienced significantly less power as much of the simple effects were not significant given our stringent criteria to avoid Type I errors. This sample size issue is associated with the next limitation.

A second limitation of the current study is the use of the C# programming language. The C# programming language is an object-oriented programming language that supports metaprogramming and is primarily an interpretative language. All of these aspects may have had an influence on the code interpretations. Different programming languages, such as C, may be perceived differently as they are not object oriented or metaprogramming languages. Third, the number of changes available for a programmer to prioritize in a given condition may have influenced the results. Participants in the transparency condition had the most opportunities to make changes in their condition. In contrast, in the reputation condition participants had the fewest number of changes they could make according to their condition. These differences were seen in the main effects of the Categories of Changes on overall *Updates* in the MANOVA. However, these influences were controlled for in the RM MANOVA when exploring the interactions. In addition, despite the ability to make more changes in the transparency condition, there was an effect such that when asked to prioritize the condition participants performed significantly more changes. This effect also occurred for performance, at least in comparison to the transparency and reputation conditions. Lastly, we did not collect any demographic data about the participants. Future research may want to collect demographic data after the code has been uploaded to explore the influence of programmer individual differences such as age or experience.

4.3. Implications and Conclusions

First, we found support for the factors found in the CTA by Alarcon et al. [13]. This research demonstrates the factors found in the CTA are viable aspects of code that influence programmer's perceptions of code, although they are not orthogonal constructs. However, greater care should be taken when operationalizing reputation in the code. Second, we interpreted our findings on prioritization in the computer sciences through the lens of multiple resource theory [4]. That is, prioritization can lead programmers to focus on certain aspects of the code while neglecting other aspects. As such, managers may want to choose their directives carefully for subordinates. Programmers that are reviewing code for a long-standing architecture may want to focus on transparency to ensure the architecture is sufficiently commented to ensure long use. In contrast, a manager that wants the flexibility and performance of the code to be inspected may want to emphasize those issues, or give little guidance as those assigned to the control condition performed relatively the same as the performance condition.

Lastly, the current study illustrates the use MTurk beyond self-reports. The MTurk platform can be utilized to collect actual behaviors, with participants unaware of the experimental nature of the task, increasing ecological validity. Granted, a drawback of the current study was not being able to collect background data such as experience and personality. However, new code allows researchers to track previous participants for longitudinal studies [41]. This code can be used to then follow up with self-reports to determine how the background data (e.g. experience, personality, etc.) relate to the performance metrics.

5. References

[1] Cybersecurity Incidents. *Office of Personnel Management*. (n.d.). Retrieved from https://www.opm.gov/cybersecurity/cybersecurity-incidents/

[2] Equifax Data Breach Affected 2.4 Million More Consumers. *Consumer Reports*. (2018, March 1). Retrieved from https://www.consumerreports.org/credit-bureaus/equifax-data-breach-was-bigger-than-previously-reported/

[3] L. Hautala, "Programmers are copying security flaws into your software, researchers warn." *CNET*. (2015, June 23). Retrieved from http://www.cnet.com/news/programmers-are-copyingsecurity-flaws-into-your-software-researchers-warn/

[4] C. D. Wickens, "Multiple resources and performance prediction," *Theoretical Issues in Ergonomics Science*, 2002, vol. 3, pp. 159-177. doi:10.1080/14639220210123806

[5] CVE Mitre. "*Heartbleed OpenSSL bug* [CVE-2014-0160]," 2014, Retrieved from https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160

[6] Z. Durumeric, J. Kasten, D. Adrian, A. Halderman, M. Bailey, F. Li, … V. Paxson, V, "The matter of Heartbleed," *Proceedings of the Internet Measurement Conference*, 2014, 14, pp. 475-488. doi:10.1145/2663716.2663755

[7] B. Grubb, "Heartbleed disclosure timeline: Who knew what and when," *Sydney Morning Herald*, 2014, Retrieved from http://www.smh.com.au/it-pro/security-it/heartbleed-disclosure-timeline-who-knew-what-andwhen-20140414-zqurk

[8] S. Henson, *GitHub*, 2014, Retrieved from https://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=96db902

[9] E. Kreft, "Heartbleed: How the net bug that caught tech experts by surprise affects you," *The Blaze*, 2014, Retrieved from http://www.theblaze.com/news/2014/04/09/heartbleed-how-the-net-bug-that-caught-tech-experts-by-surprise-affects-you/

[10] B. Sharif, M. Falcone, and J.I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," *Proceedings of the Symposium on Eye Tracking Research and Applications*, 2012, vol. 7, pp. 381-384. doi:10.1145/2168556.2168642

[11] O. Albayrak, and D. Davenport, "Impact of maintainability defects on code inspections," *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010, pp. 50-53.

[12] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The influence of non-technical factors on code review," *Proceedings of the Reverse Engineering (WCRE) 20th Working Conference on IEEE*, 2013, pp. 122-131.

[13] G. M. Alarcon, L. G. Millitello, P. Ryan, S. A. Jessup, C. S. Calhoun, and J. B. Lyons, "A descriptive model of computer code trustworthiness," *Journal of Cognitive Engineering and Decision Making*, 2017, vol. 11, pp. 107-121. doi:10.1177/1555343416657236

[14] G. M. Alarcon and T. J. Ryan, "Trustworthiness perceptions of computer code: A heuristic-systematic processing model" *Proceedings of the Hawaii International Conference on System Sciences*, 2018, pp. 5384-5393. doi: 10.24251/HICSS.2018.671

[15] D. Navon, and D. Gopher, "On the economy of the human-processing system," *Psychological Review*, 1979, vol. 86, pp. 214-255.

[16] A. Mack, and I. Rock, "Inattentional blindness: Perception without attention". In R. D. Wright (Ed.), *Visual Attention*, 1998, pp. 55-76.

[17] U. Neisser, *Cognition and reality: Principles and implications of cognitive psychology*, 1976.

[18] D. J. Simons, and C. F. Chabris, "Gorillas in our midst: Sustained inattentional blindness for dynamic events," *Perception*, 1999, vol *28*, pp. 1059-1074. doi:10.1068/P281059

[19] J. K. Caird, K. Johnston, C. Willness, M. Asbridge, M., and P. Steel, "A meta-analysis of the effects of texting on driving," *Accident Analysis and Prevention*, 2014, vol. 71, pp. 311–318. doi:10.1016/J.AAP.2014.06.005

[20] W. J. Horrey, C. D., Wickens, and K. P. Consalus, "Modeling drivers' visual attention allocation while interacting with in-vehicle technologies," *Journal of Experimental Psychology: Applied*, 2016, vol. 12, pp. 67-78. doi:10.1037/1076-898X.12.2.67

[21] G. Sperling, and M. J. Melchner, "The attention operating characteristic: Examples from visual search," *Science*, 1978 vol. 202, pp. 315-318. doi:10.1126/science.694536

[22] D. P. Brumby, N. Del Rosario, and C. P. Janssen, "When to switch? Understanding how performance tradeoffs shape dual-task strategy," *Proceedings of the International Conference on Cognitive Modeling*, 2010, pp. 19-24.

[23] A. Lang, and M. D. Basil, "Attention, resource allocation, and communication research: What do secondary task reaction times measure, anyway?" *Annals of the International Communication Association*, 1998, vol. 21, pp. 443-458. doi:10.1080/23808985.1998.11678957

[24] T. Drew, M. L. H. Võ, and J. M. Wolfe, "The invisible gorilla strikes again: Sustained inattentional blindness in expert observers," *Psychological Science*, 2013, vol. 24, pp. 1848-1853. doi:10.1177/0956797613479386

[25] J. Levy, and H. Pashler, "Task prioritization in multitasking during driving: Opportunity to abort a concurrent task does not insulate responses from dual-task slowing," *Applied Cognitive Psychology*, 2008, vol. 22, pp. 507-525. doi:10.1002/ACP.1378

[26] M. A. Regan, C. Hallett, and C. P. Gordon, "Driver distraction and driver inattention: Definition, relationship and taxonomy," *Accident Analysis & Prevention*, 2011, vol. 43, pp. 1771-1781. doi:10.1016/j.aap.2011.04.008

[27] M. J. Waller, "The timing of adaptive group responses to nonroutine events," *Academy of Management Journal*, 1999, vol. 42, pp. 127-137. doi.org/10.5465/257088

[28] M. J. Waller, N. Gupta, and R. C. Giambatista, "Effects of adaptive behaviors and shared mental models on control crew performance," *Management Science*, 2004, vol. 50, pp. 1534-1544. doi:10.1287/mnsc.1040.0210

[29] Amazon Mechanical Turk. (n.d.). Retrieved from https://www.mturk.com/https://www.mturk.com/

[30] Amazon FAQs. *Amazon Mechanical Turk*. (n.d.). Retrieved from https://www.mturk.com/mturk/help?helpPage=overview

[31] J. K. Goodman, C. E. Cryder, and A. Cheema, "Data collection in a flat world: The strengths and weaknesses of Mechanical Turk samples," *Journal of Behavioral Decision Making*, 2013, vol. 26, pp.213-224. doi:10.1002/BDM.1753

[32] G. Paolacci, and J. Chandler, "Inside the Turk: Understanding Mechanical Turk as a participant pool," *Current Directions in Psychological Science*, 2014, vol. 23, pp.184-188. doi: 10.1177/0963721414531598

[33] H. Zhou, and A. Fishbach, "The pitfall of experimenting on the web: How unattended selective attrition leads to surprising (yet false) research conclusions," *Journal of Personality and Social Psychology*, 2016, vol. 111, pp. 493-504. doi:10.1037/PSPA0000056

[34] A. J. Berinsky, G. A. Huber, and G. S. Lenz, "Evaluating online labor markets for experimental research: Amazon.com's Mechanical Turk," *Political Analysis*, 2012, vol. 20, pp.351-368. doi:10.1093/pan/mpr057

[35] M. Buhrmester, T. Kwang, and S. D. Gosling, "Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality, data?," *Perspectives on Psychological Science*, 2011, vol. 6, pp. 3-5. doi:10.1177/1745691610393980

[36] R. F. Baumeister, K. D. Vohs, and D. C. Funder, "Psychology as the science of self-reports and finger movements: Whatever happened to actual behavior?" *Perspectives on Psychological Science*, 2007, vol. *2*, pp. 396-403. doi:10.1111/j.1745-6916.2007.00051.x

[37] G. M. Alarcon, R. F. Gamble, T. J. Ryan, C. Walter, S. A. Jessup, D. W. Wood, and A. Capiola, "The influence of commenting validity, placement, and style on perceptions of computer code trustworthiness: A heuristic-systematic processing approach," *Applied Ergonomics*, 2017, vol. 70, pp. 182-193. doi:10.1016/j.apergo.2018.02.027

[38] G. M. Alarcon, R. F. Gamble, S. A. Jessup, C. Walter, T. J. Ryan, D. W. Wood, and C. S. Calhoun, "Application of the heuristic-systematic model to computer code trustworthiness: The influence of reputation and transparency," *Cogent Psychology*, 2017, Advance online publication. doi:10.1080/23311908.2017.1389640

[39] A. O. J. Cramer, D. Ravenzwaaij, D. Matzke, H. Steingroever, R. Wetzels, R. P. P. P. Grasman, … E. Wagenmakers, "Hidden multiplicity in exploratory multiway ANOVA: Prevalence and remedies," *Psychonomic Bulletin & Review*, vol. 23, pp. 640-647.

[40] R. E. Ahmed, "Software maintenance outsourcing: Issues and strategies," *Computers and Electrical Engineering*, 2006, vol. 32, pp. 449-453.

[41] L. Litman, J. Robinson, and T. Abberbock, "TurkPrime.com: A versatile crowdsourcing data acquisition platform for the behavioral sciences," *Behavior Research Methods*, 2017, vol. 49, pp. 433-442. doi:10.3758/s1342